JShelter: Give Me My Browser Back

Libor Polčák[1][a], Marek Saloň[1], Giorgio Maone[2], Radek Hranický[1][b] and Michael McMahon[3]

[1] *Brno University of Technology, Faculty of Information Technology, Božetěchova 2, 612 66 Brno, Czech Republic*

[2] *Hackademix, via Mario Rapisardi 53, 90144 Palermo, Italy*

[3] *Free Software Foundation, 51 Franklin Street Fifth Floor, MA 02110 Boston, U.S.A.*

fi

Keywords: Browser Fingerprinting, Web Privacy, Web Security, Webextension APIs, JavaScript.

Abstract: The web is used daily by billions. Even so, users are not protected from many threats by default. This paper builds on previous web privacy and security research and introduces JShelter, a webextension that fights to return the browser to users. Moreover, we introduce a library helping with common webextension development tasks and fixing loopholes. JShelter focuses on fingerprinting prevention, limitations of rich web APIs, prevention of attacks connected to timing, and learning information about the device, the browser, the user, and the surrounding physical environment and location. During the research of sensor APIs, we discovered a loophole in the sensor timestamps that lets any page observe the device boot time if sensor APIs are enabled in Chromium-based browsers. JShelter provides a fingerprinting report and other feedback that can be used by future web privacy research. Thousands of users around the world use the webextension every day.

1 INTRODUCTION

Most people interact with web pages daily. Nowadays, many activities are often carried out exclusively in a web browser, including shopping, searching for travel information, and performing leisure activities. For several years, browser vendors have been adding new JavaScript APIs to solicit the development of rich web applications (Snyder et al., 2016).

Consequently, web visitors face several threats like hostile tracking (Matte et al., 2020; ICO, 2019; APD, 2022), fingerprinting (Laperdrix et al., 2020; Iqbal et al., 2021), and malware (Bergbom, 2019).

This paper presents JShelter, a web browser extension (webextension) that allows users to tweak the browser APIs. Additionally, JShelter detects and prevents fingerprinting. Moreover, JShelter blocks attempts to misuse the browser as a proxy to access the local network. JShelter educates users by explaining fingerprinting APIs in a report. JShelter integrates several previous research projects like Chrome Zero (Michael Schwarz and Gruss, 2018) and little-lies-based fingerprinting prevention (Nikiforakis et al., 2015; Pierre Laperdrix, 2017). As current webextension APIs lack a reliable way to modify JavaScript APIs in different contexts like iframes and web workers, we needed to solve the reliable injection. This paper introduces NoScript Commons Library (NSCL)[1] that other privacy- and security-related webextensions can reuse to solve common tasks like the reliable injection of JavaScript code into the page JavaScript context before the page scripts can access the context. We implemented JShelter for Firefox and Chromium-based browsers like Chrome, Opera, and Edge.

This paper is organised as follows. Section 2 overviews related work and specifies the threat model that we adopted. Section 3 provides the design decisions. Section 4 evaluates the JShelter features. Section 5 concludes this paper.

2 THREATS AND RELATED WORK

JShelter focuses on threats that affect the mainstream population. The considered adversary attacks or derives information in a way that works in mainstream browsers. The attacker focuses on these browsers and attacks that are light on performance.

[a] https://orcid.org/0000-0001-9177-3073

[b] https://orcid.org/0000-0001-6315-8137

[1] https://noscript.net/commons-library

PolÄ■Ãąk, L., SaloÅĹ, M., Maone, G., HranickÃ¡, R. and McMahon, M.
JShelter: Give Me My Browser Back.
DOI: 10.5220/0011965600003555
In *Proceedings of the 20th International Conference on Security and Cryptography (SECRYPT 2023)*, pages 287-294
ISBN: 978-989-758-666-8; ISSN: 2184-7711
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

Threat 1 (T1): User Tracking. is a threat to fundamental rights identified by both academia (Matte et al., 2020) and European data protection authorities (APD, 2022; ICO, 2019). Historically, trackers stored user identifiers in third-party cookies. As browser vendors limit third-party cookies, trackers move to alternative ways of identifying users, like browser fingerprinting (Laperdrix et al., 2020).

JShelter protects users from tracking by modifying the results of the APIs used by fingerprinters and other actors trying to uniquely identify users. To mitigate browser fingerprinting, JShelter implements a technique employed by Brave browser that modifies API differently on different domains and in different sessions. The goal is to create a unique fingerprint for each domain and session. Such fingerprints cannot be used for cross-domain linking of the same browser. Additionally, JShelter can detect and block fingerprinting attempts, as explained in Section 3.1.

Threat 2 (T2): Very Rich Browser APIs. Web pages can communicate with the web browser and the underlying operating system through APIs supporting video calls, audio and video editing, navigation, and augmented and virtual reality. Nevertheless, most web pages do not need these advanced APIs (Snyder et al., 2017). Service Worker API allows Man-in-the-Middle adversaries inject long-lasting trackers (Polčák and Jeřábek, 2023).

Webextensions like NoScript Security Suite and uMatrix Origin allow users to block resources, including JavaScript code, based on their domain. Nevertheless, malicious code may be only a part of a resource; the rest of the resource can be necessary for correct page functionality. In contrast, JShelter blocks specific calls. Hence, other code can render the page as expected, and only dangerous APIs are affected.

Threat 3 (T3): Local Network Scanning. A web page can try to exploit the web browser as a proxy between the remote website and resources in the local network. (Bergbom, 2019) demonstrated that executing arbitrary commands on a local machine is possible under certain circumstances (in this case, it was an insecure Jenkins configuration). Subsection 4.3 explains that JShelter mitigates the possibility of exploiting the browser as a proxy to a local network.

Threat 4 (T4): Microarchitectural Attacks. Previous research also focused on side-channel attacks that can reveal what the user has recently done with the device. For example, content-based page deduplication performed by an operating system or a virtual machine hypervisor can reveal if specific images or websites are currently opened (Gruss et al., 2015) on the same device (hardware), possibly on another virtual machine. JShelter modifies timestamps in all APIs to make attacks requiring precise time measurements harder.

3 DESIGN DECISIONS

This section covers the design decisions of JShelter and the countermeasures we decided to implement.

JShelter consists of (1) JavaScript Shield (JSS) that modifies or disables JavaScript APIs, (2) Fingerprint Detector (FPD) that provides heuristic analysis of fingerprinting behaviour, and (3) Network Boundary Shield (NBS) that detects attempts to misuse the browser as a proxy to the local network (T3).

3.1 Fingerprint Detector

Fingerprint Detector (FPD) monitors APIs that are commonly used by fingerprinters and applies a heuristic approach to detect fingerprinting behaviour in real-time (see threat T1). When a fingerprinting attempt is detected, FPD notifies the user. The user can configure JShelter to reactively block subsequent asynchronous HTTP requests initiated by the fingerprinting page and clear the storage facilities where the page could have stored a (partial) fingerprint. However, this behaviour may break the page. The goal of the aggressive mode is to prevent the page from uploading the full fingerprint to a server. However, the fingerprinter can gradually upload detected values, and a partial fingerprint can leak from the browser.

The heuristics are based on prior studies (Laperdrix et al., 2020; Englehardt and Narayanan, 2016; Iqbal et al., 2021) that proved it to be a viable approach with a very low false-positive rate. FPD counts calls of JavaScript API endpoints known to be used for fingerprinting (Iqbal et al., 2021; Fietkau et al., 2021)[2].

FPD is not based on code analyses, so it overcomes any obfuscation of fingerprinting scripts.

FPD provides a report that summarises FPD findings on the visited web page, see Figure 1. The report aims to educate users about fingerprinting and clarifies why FPD notified the user and optionally blocked the page. As the report can be generated from passive observation of a web page (no API blocking), we

[2] Including fingerprinting tools like FingerprintJS, https://github.com/fingerprintjs, Am I Unique, https://amiunique.org/, and Cover Your Tracks, https://coveryourtracks.eff.org/. Furthermore, we analysed FPMON (Fietkau et al., 2021) and DFPM https://github.com/freethenation/DFPM

expect that other researchers will use passive FPD to study fingerprinting in more detail.

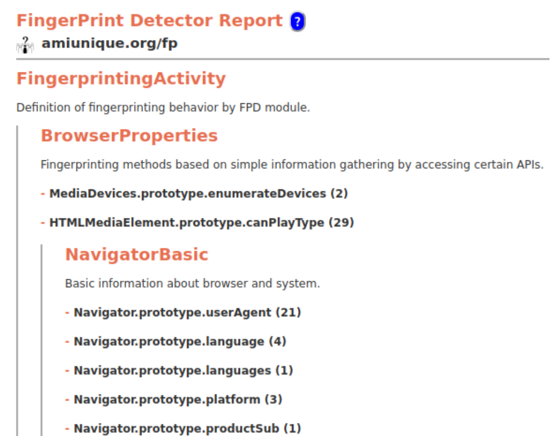


Figure 1: An excerpt from an FPD report on AmIUnique.org. A user sees what APIs the visited page called.

3.2 JavaScript Shield

JSS focuses on timestamp spoofing (threat T1 and T4), fingerprint modifications (threat T1) and disabling APIs available to visited pages (threat T2).

JShelter currently modifies 113 APIs, which include APIs considered by previous works (Michael Schwarz and Gruss, 2018; Iqbal et al., 2021; Snyder et al., 2017) and APIs that Apple declined to implement. For each API, we decide its relevance on an individual basis. Usually, we do not modify APIs already explicitly permitted by the user. However, the analysis might provide an example where the user still wants to limit the precision of the API. For example, Geolocation API allows the page to learn a very precise location while the user might be interested in services in the city. Hence, JShelter allows fine-tuning the precision of the Geolocation (and other APIs).

Additionally, the slightest mismatch between the results of two APIs can make the user more visible to fingerprinters (Laperdrix et al., 2020). Hence, we consider each protection that we decide to implement in JShelter from the point of fingerprintability, the threat of leaking information about the browser or user and other threats presented in Section 2. JShelter tries to mimic a stationary device with consistent and plausible readings.

JSS provides a profile that focuses on making the browser appear differently to distinct fingerprinting origins by slightly modifying the results of API calls (little lies) (Nikiforakis et al., 2015; Pierre Laperdrix, 2017). The little lies approach builds on the Farbling protection implemented in Brave[3] and applies the same or very similar protection. These little lies result in different websites calculating different fingerprints. Moreover, a previously visited website calculates a different fingerprint in a new browsing session. Consequently, cross-site tracking is more complicated.

Another profile focuses on limiting the information provided by the browser by returning fake values from the protected APIs. Some are blocked completely, some provide meaningful but rare values, and others return meaningless values. This level makes the user fingerprintable because the results of API calls are generally modified in the same way on all websites and in each session.

3.2.1 Interaction Between JavaScript Shield and Fingerprint Detector

Both JSS and FPD aim to prevent fingerprinting. Both are necessary for JShelter.

The blocking mode of FPD breaks pages. Users are typically tempted to access the content even when they know they are being fingerprinted. Consequently, they turn FPD off for such pages. JSS ensures that these users are not linkable across origins and sessions.

The JSS profile focusing on limiting information access will likely result in the same fingerprint for all domains; hence, we strongly advise users of this profile to activate FPD.

We expect most users to stick with the default profile creating little lies. Future research should validate the current approach. For example, JShelter and Brave create indistinguishable changes to canvas readings. These are sufficient for a fingerprinter that creates a hash of the readings. Nevertheless, an advanced fingerprinter might, for example, read the colours of specific pixels to determine a presence of a font (different fonts produce a different pixel-wise-long output of the same text). As both Brave and JShelter modify only the least significant bit of each colour, the fingerprinter can ignore this bit and get the information on installed fonts. Hence, FPD is beneficial as it offers additional protections.

3.2.2 Sensors

JShelter tries to simulate a stationary device and consequently completely spoofs the readings of AmbientLight, AbsoluteOrientation, RelativeOrientation, Accelerometer, LinearAcceleration, Gravity, Gyroscope,

[3]See https://github.com/brave/brave-browser/issues/8787 and https://github.com/brave/brave-browser/issues/11770

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

and Magnetometer sensors. JShelter also spoofs Geolocation API that can be either completely blocked or return a modified location derived from the reading from the original API.

Instead of using the original data, JShelter returns artificially generated values that look like actual sensor readings. Hence the spoofed readings fluctuate around a value that is unique per origin and session.

We observed sensor readings from several devices to learn the fluctuations of stationary devices in different environments. Most of the sensors have small deviations. However, magnetometer fluctuates heavily. JShelter simulates the fluctuations by adding multiple sines for each axis. Each sine has a unique amplitude, phase shift, and period. The number of sines per axis is chosen pseudorandomly. JShelter currently employs 20 to 30 sines for each axis. Nevertheless, the optimal configuration is subject to future research. More sines give less predictable results at the cost of increased computing complexity.

3.2.3 User in Control

The number of modified APIs is high. We expect that users will encounter pages broken by JShelter or that do not work as expected. For example, the user might want to play games with a gamepad device on some pages or make a call on others.

JSS allows each user to fine-tune the protection for each origin. Some users reported that they would prefer to avoid digging into the configuration. Those can disable JSS for the domain with a simple ON/OFF popup switch. More experienced users can react to information provided by FPD and turn off JSS fingerprint protection when the visited site does not behave as a fingerprinter. The most experienced users can fine-tune the behaviour per API group. Figure 2 shows an example of a user accessing a page that allows video calls. The user sees the groups with APIs that have been called by the visited page at the top and can quickly fix a broken page.

Figure 2: JSS reports back which APIs are being used by the page.

3.3 Effective Modifications of the JavaScript Environment

Both JSS and FPD depend on replacing (wrapping) of the built-in JavaScript APIs and built-in object behaviour. JShelter employs the same mechanism proposed by (Michael Schwarz and Gruss, 2018) in Chrome Zero. However, Chrome Zero was a proof-of-concept with no modification in the last four years. (Shusterman et al., 2021) identified several problems with Chrome Zero:

1. Unprotected prototype chains (*issue 1*): the original implementation is available through the prototype chain because Chrome Zero protects a wrong property.
2. Delayed JavaScript environment initialisation (*issue 2*): Current webextension APIs lack a reliable and straightforward way to inject scripts modifying the JavaScript environment before page scripts start running. JShelter and Chrome Zero allow configurable protection that may differ per origin, so they need to load the configuration during each page load. Hence, a naïve implementation with asynchronous APIs may allow page scripts to access original, unprotected API calls. Note that once page scripts can access the original API implementation, they can store the unprotected version. A webextension cannot reverse the leak.
3. Missed context (*issue 3*): Chrome Zero does not apply protection in iframes and worker threads.

In addition, Firefox suffers from a long-standing unfixed bug (Mozilla Bugzilla, 2016) that prevents Firefox webextensions from working correctly on pages whose Content Security Policy (CSP) forbids inline scripts (*issue 4*).

JShelter tackles issue 1 in two steps. (1) Developers analyse the prototype chain and pick the correct object implementing the property or method to wrap. (2) The injection code checks at runtime the correct position to apply the wrapper.

To overcome issues 2–4, we needed to develop a reliable cross-browser early script injection. As the same issues affect several privacy and security webextensions, we refactored the code from NoScript Security Suite into NSCL and made it publicly available for reusing and contributing back.

NSCL abstracts common functionality shared among security and privacy webextensions to minimise the development and maintenance burden on webextension maintainers. For example, an adversary can access an API through the window object, an iframe, or a worker. A webextension modifying the API needs to modify each possibility. By modifying only some ways to access the API, the webextension not only gives an attacker the possibility to learn original values offered by the API but also reveals that the browser behaves strangely. Additionally, NSCL provides consistent implementation across multiple

browser engines. Hence, developers do not need to study browser-dependent implementation details.

NSCL tackles issue 2 by preprocessing URL-dependent configuration inside a `BeforeNavigate` event handler that has access to the destination URL and JShelter can build a configuration object in advance and have it ready during the document start event (before any page script can run). However, due to race conditions, when the configuration object is missing in the document start event, NSCL provides `SyncMessage` API to retrieve the correct settings before it is interleaved with concurrent scripts.

To address issue 3, `manifest.json` (the configuration of the webextension) registers code injection into all the newly created windows, including subframes. Unfortunately, `window.open()`, `contentWindow`, and `contentDocument.window` allow access to a new window object immediately after its creation (synchronously) before any initialisation (including the injection registered in `manifest.json`) occurs. NSCL wraps the affected calls to recursively wrap the newly created window just before the window is accessible to page scripts.

A further possibility to access unwrapped APIs are subframe windows of all kinds, also immediately available at creation time by indexing their parent window as an unwrappable pseudo array (e.g. `window[0]` is a synonym of `window.frames[0]`). NSCL automatically patches all not yet patched `window[n]` objects every time the DOM structure is modified, potentially creating new windows. This requires that NSCL wraps all methods and accessors by which the DOM can be changed in JavaScript.

Regarding web workers, JShelter disables them. NSCL provides another option: wrapping workers by injecting the wrappers in their own browser context via its `patchWorkers()` API.

Finally, NSCL works around issue 4 by leveraging a Firefox-specific privileged API meant to safely share functions and objects between page scripts and WebExtensions[4].

4 EVALUATION

This section evaluates the different JShelter parts.

[4]https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Sharing_objects_with_page_scripts

4.1 JavaScript Shield

4.1.1 Fingerprinting Inconsistencies

Besides a few bugs that we intend to fix, we are aware that a fingerprinter may observe some inconsistencies. For example, JShelter modifies each read canvas. Should the page scripts probe a single-colour-filled canvas, JShelter would introduce small changes in some pixels. Hence, a page script might learn that protection against canvas fingerprinting is in place.

The little lies modifications (see Section 3.2) have a performance hit. For all APIs that allow obtaining hardware-rendered data like the Canvas, WebGL, and WebAudio APIs, JShelter needs to access all data in two iterations, first to create a hash that controls the modifications in the second iteration. Hence, the same content is deterministically modified the same way, and different content is modified differently.

`AudioBuffer.prototype.getChannelData` allows quick access to pulse-code modulation audio buffer data without data copy. A fingerprinter might be interested in a couple of samples only. However, the spoofing mechanism needs to access all data, so the method is much slower (learning that the time of `getChannelData` takes too long is usable for fingerprinting).

We are not aware of any isolated side-effect that reveals JShelter. For example, page scripts can detect some similar webextensions by calling `Function.prototype.toString` for the modified APIs. Should `toString` return the wrapping code modifying the API rather than the original value, it might reveal a unique text as other webextensions modifying the same API call by the same technique will likely use a different code. Nevertheless, we are aware and do not hide that users of JShelter are vulnerable to focused attacks. Our goal is to offer protections indistinguishable from another privacy-improving tool for each modified API. Nevertheless, a focused observer will very likely be always able to learn that a user is using JShelter if they aggregate the observable inconsistencies of all APIs produced by JShelter.

4.1.2 Timing Events

JShelter implements rounding and afterwards, by default, randomises the timestamps as Chrome Zero does (Michael Schwarz and Gruss, 2018). In comparison, Firefox Fingerprinting Protection and Tor Browser implement only rounding, which makes the technique visually easily detectable. Compared with Chrome Zero, JShelter modifies all APIs that produce

timestamps, including events (see threat T1), geolocation, gamepads, virtual reality and sensors.

4.1.3 Sensor Timestamp Loophole

We discovered a loophole in the `Sensor.timestamp` attribute[5]. The value describes when the last `Sensor.onreading` event occurred in millisecond precision. We observed that the reported time is the time since the last boot of the device. Exposing such information is dangerous as it allows fingerprinting the user easily as devices boot at different times.

JShelter protects the device by provisioning the time since the browser created the page context (the same value as returned by `performance.now()`. Such timestamps uniquely identify the reading without leaking anything about a device. Future work can determine if such behaviour appears in the wild. If all devices and browsers incorporate the loophole, we should provide a random boot time.

4.1.4 Fake Magnetometer Evaluation

Figure 3 shows readings from a real and fake magnetometer. The left part (a) shows a stationary device. The magnetic field is not stable due to small changes in Earth's magnetic field and other noise. The middle part of the figure (b) shows a device that changed its position several times during the measurement.

Figure 3 (c) shows readings generated by JShelter fake magnetometer. The values look like actual sensor readings. Nevertheless, the generator uses a series of constants whose optimal values should be the subject of future research and improvements.

4.2 Fingerprint Detector Effectivity

The FPD heuristics were designed to keep the number of false positives as low as possible. As FPD can optionally block all subsequent requests by a fingerprinting page and JShelter provides complementary protections, FPD blocks only indisputable fingerprinting attempts. We conducted real-world testing of FPD and refined its detection heuristics accordingly.

Regarding testing methodology, we manually visited homepages and login pages of the top 100 websites from the Tranco list[6]. We randomly replaced inaccessible websites by websites from the top 200 list. Before visiting a website, we wiped browser caches and storage to remove previously-stored identifiers. Hence, the visited pages may have deployed fingerprinting scripts more aggressively to identify the user and reinstall the identifier.

To boost the probability of fingerprinting even more, we switched off all protection mechanisms offered by the browser. However, we blocked third-party cookies because our previous experience suggests that the missing possibility to store a permanent identifier tempts trackers to start fingerprinting. We repeated the visits with both Google Chrome and Mozilla Firefox.

We used FPMON (Fietkau et al., 2021), DFPM[7], and JShelter to find the ground truth. For each visited page, we computed its fingerprinting score. FPMON reports fingerprinting pages with colour. We assigned yellow colour 1 point and red colour 3 points. DFPM reports danger warnings. If DFPM reports one danger warning, we assign 1 point to the page. For a higher number of danger warnings, we assign 3 points to the page. Therefore, each page gets a fingerprinting score from 0 to 6. We consider each page with the score of 6 or 4 to engage in fingerprinting. Additionally, we inspected pages with the score lower than 4 flagged by FPD. We detected five additional fingerprinting pages after manual inspection.

Table 1 shows the accuracy and the sum of true positives and true negatives of the tested tools. In total, we tested 98 home pages and 81 login pages; 2 home pages are actually login pages, we removed duplicate login pages, and some sites do not have a login page. JShelter is more accurate in fingerprinting detection when compared with the scenario when FPMON and DFPM have low confidence in fingerprinting detection (they score 1 point). JShelter is slightly worse compared to the scenario in which the other tools are confident that they detected fingerprinting. The differently evaluated pages are typically borderline cases. For example, JShelter does not detect fingerprinting on Google and Facebook login pages, while both FPMON and DFPM detect fingerprinting. As the number of accessed APIs is not high and users would likely turn FPD off for these pages, we do not intend to modify FPD heuristics.

4.3 Network Boundary Shield

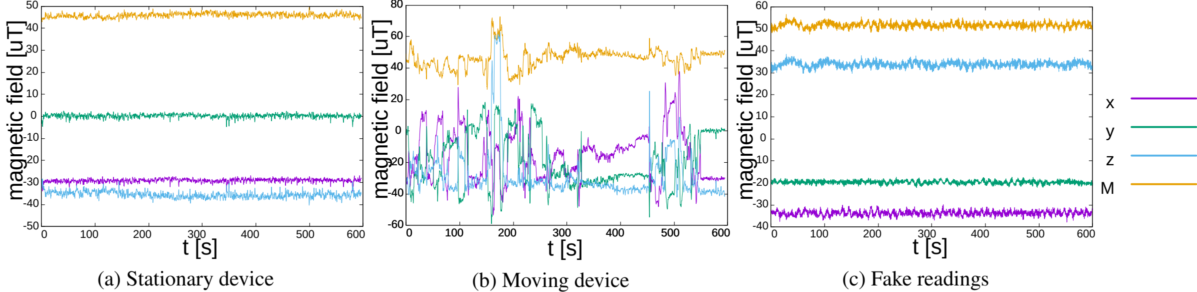
4.3.1 Localhost Scanning

Some web pages, like ebay.com, scan (some users) for open local TCP ports to detect bots with open remote desktop access or possibly to create a fingerprint. The

[5]Tested with Samsung Galaxy S21 Ultra; Android 11, kernel 5.4.6-215566388-abG99BXXU3AUE1, Build/RP1A.200720.012.G998BXXU3AUE1, Chrome 94.0.4606.71 and Kiwi (Chromium) 94.0.4606.56 and Xiaomi Redmi Note 5; Android 9, kernel 4.4.156-perf+, Build/9 PKQ1.180901.001, Chrome 94.0.4606.71

[6]https://tranco-list.eu/list/23W9/1000000

[7]https://github.com/freethenation/DFPM



(a) Stationary device (b) Moving device (c) Fake readings

Figure 3: Magnetometer readings.

Table 1: Fingerprint detection accuracy of tested webextensions based on the manual crawl of the top 100 web pages according to the Tranco list.

		Home pages	Login pages
JShelter	Detected	96 (98.0 %)	77 (95.1 %)
FPMON	red	79 (80.6 %)	66 (81.5 %)
	red/yellow	96 (98.0 %)	80 (98.8 %)
DFPM	2+ dangers	70 (71.4 %)	66 (81.5 %)
	1+ dangers	98 (100 %)	81 (100 %)

web page instructs the browser to connect to the *localhost* (`127.0.0.1`) and monitors the errors to detect if the port is opened or closed.

When we developed NBS we did not anticipate localhost port scanning. When we first encountered the eBay port scanning case, we knew that this behaviour should trigger NBS as the requests cross network boundaries. We accessed ebay.com, detected the scanning by Web Developer Tools and checked that NBS is indeed triggered and works as expected.

4.3.2 Comparison with Private Network Access

Recently, Google announced Private Network Access (PNA)[8] that should become W3C standard[9]. PNA solves the same problem as NBS, but the solution is different. PNA-compatible browsers send HTTP Requests to the local networks with the additional header: *Access-Control-Request-Private-Network: true*.

The local resource can allow such access with HTTP reply header *Access-Control-Allow-Private-Network: true*. If it does not, the browser blocks the access.

NBS works differently. Firefox version leverages DNS API to learn that a public web page tries to access the local network and blocks the request before the browser sends any data. Chromium-based browsers do not support DNS API, so the first request goes through. NBS learns the IP address during the reply processing. NBS blocks any future request before it is made once it learns the IP address during the reply processing. Hence, NBS limits the network bandwidth and prevents any state modification on a local node that may be caused by a request going through, except for the learning phase in Chromium-based browsers. Both approaches solve threat T3; it is up to the user what solution they prefer.

Note that Google plans to fully deploy Chrome PNA in version 113, so Chrome users without JShelter or another webextension with similar capabilities are not protected at the time of the writing of this paper[10].

5 CONCLUSION

Previous research established that browser security, privacy, and customizability are important topics (Laperdrix et al., 2020; Michael Schwarz and Gruss, 2018; Bergbom, 2019). The imminent danger of third-party cookie removal forces trackers to employ even more privacy-invading techniques. Real-time bidding leaves users easy targets for various attacks, including gaining information about other applications running on the local computer (Gruss et al., 2015). Moreover, continuous additions of new JavaScript APIs open new ways for fingerprinting the browsers and gaining additional knowledge about the browser or user preferences and physical environment. One of the major concerns is a need for more effective tools that everyday user wants to use. Current methods to tackle web threats are list-based blockers that might be evaded with a change of URL, specialised browsers, or research-only projects that are quickly abandoned.

In contrast, JShelter is a webextension that can be installed on major browsers and does not require the user to change the browser and routines. We integrate and improve several previous research projects like Chrome Zero (Michael Schwarz and

[8]https://developer.chrome.com/blog/private-network-access-prefilght/

[9]https://wicg.github.io/private-network-access/

[10]https://chromestatus.com/feature/5737414355058688

Gruss, 2018), little-lies-based fingerprinting prevention (Nikiforakis et al., 2015; Pierre Laperdrix, 2017), and ideas for limiting APIs brought by Web API Manager (Snyder et al., 2017). JShelter comes with a heuristic-based fingerprint detector and prevents web pages from misusing the browser as a proxy to access the local network and computer. We solved issues with reliable environment modifications that stem from insufficient webextension APIs that open many loopholes that previous research exploited (Shusterman et al., 2021). In addition to JShelter, we introduced NSCL. Both NoScript Security Suite and JShelter include NSCL. Moreover, NSCL is available for other privacy- and security-related webextensions.

In cooperation with Free Software Foundation, we aim for long-term JShelter development; thus, users' privacy and security should be improved in the future.

ACKNOWLEDGEMENTS

This project was funded through the NGI0 PET Fund, a fund established by NLnet with financial support from the European Commission's Next Generation Internet programme, under the aegis of DG Communications Networks, Content and Technology under grant agreement No 825310 as JavaScript Restrictor and JShelter projects. This work was supported in part by the Brno University of Technology grant Smart information technology for a resilient society (FIT-S-23-8209).

REFERENCES

APD (2022). Decision on the merits 21/2022 of 2 February 2022. APD — Autorité de protection des données. Available online at https://www.autoriteprotectiondonnees.be/publications/decision-quant-au-fond-n-21-2022-english.pdf, unofficial translation from Dutch.

Bergbom, J. (2019). Attacking the internal network from the public internet using a browser as a proxy. Forcepoint research report available at https://www.forcepoint.com/sites/default/files/resources/files/report-attacking-internal-network-en_0.pdf.

Englehardt, S. and Narayanan, A. (2016). Online tracking: A 1-million-site measurement and analysis. In *CCS '16*, pages 1388–1401.

Fietkau, J., Thimmaraju, K., Kybranz, F., Neef, S., and Seifert, J.-P. (2021). The elephant in the background: A quantitative approach to empower users against web browser fingerprinting. In *WPES '21*, page 167–180.

Gruss, D., Bidner, D., and Mangard, S. (2015). Practical memory deduplication attacks in sandboxed Javascript. In *Computer Security – ESORICS 2015*, pages 108–122. Springer International Publishing.

ICO (2019). Update report into adtech and real time bidding. ICO — Information Commissioner's Office. Available online at https://ico.org.uk/media/about-the-ico/documents/2615156/adtech-real-time-bidding-report-201906.pdf.

Iqbal, U., Englehardt, S., and Shafiq, Z. (2021). Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In *IEEE Symposium on Security & Privacy*, pages 1143–1161.

Laperdrix, P., Bielova, N., Baudry, B., and Avoine, G. (2020). Browser fingerprinting: A survey. volume 14. ACM.

Matte, C., Bielova, N., and Santos, C. (2020). Do cookie banners respect my choice? Measuring legal compliance of banners from IAB Europe's Transparency and Consent Framework. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 791–809.

Michael Schwarz, M. L. and Gruss, D. (2018). Javascript zero: Real javascript and zero side-channel attacks. In *NDSSS 2018*.

Mozilla Bugzilla (2016). [meta] Page CSP should not apply to content inserted by content scripts (v2 issue). Available online at https://bugzilla.mozilla.org/show_bug.cgi?id=1267027.

Nikiforakis, N., Joosen, W., and Livshits, B. (2015). PriVaricator: Deceiving fingerprinters with little white lies. In *WWW '15*, pages 820—-830.

Pierre Laperdrix, Benoit Baudry, V. M. (2017). FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *9th International Symposium on Engineering Secure Software and Systems*, page 17.

Polčák, L. and Jeřábek, K. (2023). Data protection and security issues with Network Error Logging. In *Proceedings of the 20th International Conference on Security and Cryptography*. SciTePress - Science and Technology Publications.

Shusterman, A., Agarwal, A., O'Connell, S., Genkin, D., Oren, Y., and Yarom, Y. (2021). Prime+Probe 1, JavaScript 0: Overcoming browser-based Side-Channel defenses. In *USENIX Security 21*, pages 2863–2880.

Snyder, P., Ansari, L., Taylor, C., and Kanich, C. (2016). Browser feature usage on the modern web. In *IMC '16*, pages 97–110.

Snyder, P., Taylor, C., and Kanich, C. (2017). Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *CCS '17*, pages 179–194.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS